(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: **0 546 682 A2**

(12) EUROPEAN PATENT APPLICATION

(21) Application number: **92310240.4**

(51) Int. Cl.5: **G06F 9/44**

(22) Date of filing: **09.11.92**

(30) Priority: **12.12.91 US 805777**

(43) Date of publication of application:
16.06.93 Bulletin 93/24

(84) Designated Contracting States:
DE FR GB

(71) Applicant: **International Business Machines Corporation**
Old Orchard Road
Armonk, N.Y. 10504(US)

(72) Inventor: **Conner, Mike Haden**
4416 Walhill Lane
Austin, Texas 78759(US)
Inventor: **Martin, Andrew Richard**
1070 Mearns Meadow No. 534
Austin, Texas 78758(US)
Inventor: **Raper, Larry Keith**
7860 Lakewood Drive
Austin, Texas 78750(US)

(74) Representative: **Blakemore, Frederick Norman**
IBM United Kingdom Limited Intellectual Property Department Hursley Park
Winchester Hampshire SO21 2JN (GB)

(54) Parent class shadowing.

(57) A method, system and program for supporting a dynamic bind between a derived class and its parent class. A processor provides for the registration of class objects and dynamic binding of derived class objects to their parent class objects based on the registration mechanism. The SOM object model removes static references to class objects by having all the parent class information available at runtime through the parent class object. Thus, when the derived class implementation needs information about the size of the parent class state data structure, the addresses of the parent class method procedures, or access to the parent class method procedure table the appropriate information is retrieved from the parent class object.

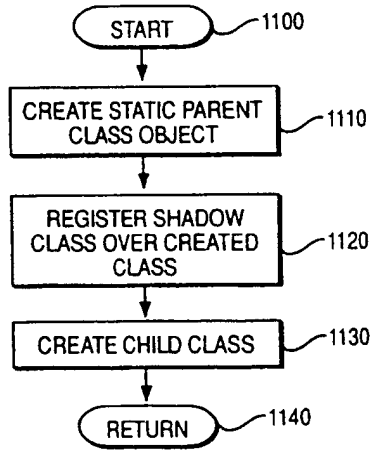


FIG. 11

EP 0 546 682 A2

default class management. Hence, a compiled software environment that requires the ability to "re-parent" a class can achieve this by subclassing the SOM class manager, substituting a new class manager instance, and enforcing new rules for locating classes and resolving relationships between them.

Matter forming the subject of the present patent specification also forms the subject of our European patent applications claiming priority from U.S. patent Applications 07/805,778 and 07/805,668 filed on 12 December 1992.

Brief Description of the Drawings

Figure **1** is a block diagram of a personal computer system in accordance with the subject invention;
Figure **2** is a drawing of a SOM data structure in accordance with the subject invention;
Figure **3** is a drawing of a SOM class data structure in accordance with the subject invention;
Figure **4** is a flowchart depicting a language neutral object interface in accordance with the subject invention;
Figure **5** is a flowchart depicting a link, load and execution of an application using SOM objects in accordance with the subject invention;
Figure **6** is a flowchart depicting the creation of a new SOM class in accordance with the subject invention;
Figure **7** is a flowchart depicting the detailed construction of a new SOM class in accordance with the subject invention;
Figure **8** is a flowchart depicting the detailed construction of a new SOM generic class object in accordance with the subject invention;
Figure **9** is a flowchart depicting the detailed initialization of a new SOM class object in accordance with the subject invention;
Figure **10** is a flowchart depicting the detailed initialization of a SOM class data structure with offset values in accordance with the subject invention;
Figure **11** is a flowchart depicting the detailed parent class shadowing of a statically defined class hierarchies in accordance with the subject invention;
Figure **12** is a flow diagram depicting the redispatch method in accordance with the subject invention;
Figure **13** is a flowchart depicting the detailed initialization of the offset value in a SOM class data structure for a single public instance variable in accordance with the subject invention;
Figure **14** is a flowchart depicting the detailed control flow that occurs when a redispatch stub is employed to convert a static method call into a dynamic method call in accordance with the subject invention; and
Figure **15** is a flowchart depicting the detailed control flow that initialize a method procedure table for a class in accordance with the subject invention.

Detailed Description Of The Invention

The invention is preferably practiced in the context of an operating system resident on an IBM PS/2 computer available from IBM Corporation. A representative hardware environment is depicted in Figure **1**, which illustrates a typical hardware configuration of a workstation in accordance with the subject invention having a central processing unit **10**, such as a conventional microprocessor, and a number of other units interconnected via a system bus **12**. The workstation shown in Figure **1** includes a Random Access Memory (RAM) **14**, Read Only Memory (ROM) **16**, an I/O adapter **18** for connecting peripheral devices such as disk units **20** to the bus, a user interface adapter **22** for connecting a keyboard **24**, a mouse **26**, a speaker **28**, a microphone **32**, and/or other user interface devices such as a touch screen device (not shown) to the bus, a communication adapter **34** for connecting the workstation to a data processing network and a display adapter **36** for connecting the bus to a display device **38**. The workstation has resident thereon the OS/2 base operating system and the computer software making up this invention which is included as a toolkit.

Object-Oriented Programming is quickly establishing itself as an important methodology in developing high quality, reusable code. The invention includes a new system for developing class libraries and Object-Oriented programs. This system is called the system Object Model (SOM). A detailed description of object oriented programming, SOM, and a comparison to other object-oriented languages is provided to aid in understanding the invention.

Most C programmers can imagine how such functions would be written. The ⟨push()⟩ function, for example, appears below.

```
void push(Stack *thisStack, void *nextElement)
{
  thisStack->stackArray[thisStack->stackTop] = nextElement;
  thisStack->stackTop++;
}
```

A client program might use this stack to, say, create a stack of words needing interpretation:

```
main()
{
  Stack *wordStack;
  char *subject = "Emily";
  char *verb = "eats";
  char *object = "ice cream";
  char *nextWord;
  wordStack = create();
  push(wordStack, object);
  push(wordStack, verb);
  push(wordStack, subject);

  /* ... */
  while (nextWord = pop(wordStack)) {
    printf("%s\n", nextWord);
    /* ... */
  }
}
```

This example can be used to review Object-Oriented Programming. A class is a definition of an object. The definition includes the data elements of the object and the methods it supports. A ⟨stack⟩ is an example of a class. A stack contains two data elements (⟨stackArray⟩ and ⟨stackTop⟩), and supports three methods, ⟨create()⟩, ⟨push()⟩, and ⟨pop()⟩. A method is like a function, but is designed to operate on an object of a particular class. An object is a specific instance, or instantiation, of a class. The object ⟨wordStack⟩ is an object of class ⟨Stack⟩, or ⟨wordStack⟩ is an instance of a stack.

Every method requires a specific object on which it is to operate. This object is called a target object, or sometimes a receiving object. Notice that each method (except ⟨create()⟩) takes as its first parameter a pointer to the target object. This is because a program may have many objects of a given class, and each are potential targets for a class method.

There are three important advantages of this type of organization. First, generic concepts are developed which can be reused in other situations in which similar concepts are appropriate. Second, self-contained code is developed, which can be fully tested before it is folded into our program. Third, encapsulated code is developed in which the internal details are hidden and of no interest to the client. A client ⟨main()⟩ program need know nothing about the ⟨Stack⟩ class other than its name, the methods it supports, and the interfaces to these methods.

Inheritance introduces some additional semantics. A specialized class is said to be derived from a more generalized class. The general class is called the parent class, or sometimes, the base class. The specialized class is called the child class, or sometimes, the derived class. A child class is said to inherit the characteristics of its parent class, meaning that any methods defined for a parent are automatically defined for a child. Thus, because ⟨GraduateStudent⟩ and ⟨UnderGraduateStudent⟩ are both derived from ⟨Student⟩, they both automatically acquire any methods declared in their common parent.

Override resolution refers to invoked methods being resolved based not only on the name of the method, but also on a class place within a class hierarchy. This allows us to redefine methods as we derive classes. We might define a ⟨printStudentInfo()⟩ method for ⟨Student⟩ and then override, or redefine, the method in both ⟨UnderGraduateStudent⟩, and ⟨GraduateStudent⟩. Override resolution resolves based on the type of the target object. If the target object type is a ⟨Student⟩, the ⟨Student⟩ version of ⟨printStudentInfo()⟩ is invoked. If the target object type is a ⟨GraduateStudent⟩, the ⟨GraduateStudent⟩ version of ⟨printStudentInfo()⟩ is invoked.

DEFINING CLASSES IN SOM

The process of creating class libraries in SOM is a three step process. The class designer defines the class interface, implements the class methods, and finally loads the resulting object code into a class library. Clients either use these classes directly, make modifications to suit their specific purposes, or add entirely new classes of their own.

In SOM, a class is defined by creating a class definition file. The class definition file is named with an extension of "csc". In its most basic form, the class definition file is divided into the following sections:

1. Include section

This section declares files which need to be included, much like the C ⟨#include⟩ directive.

2. Class name and options

This section defines the name of the class and declares various options.

3. Parent information

This defines the parent, or base, class for this class. All classes must have a parent. If a class is not derived from any existing classes, then it's parent will be the SOM defined class ⟨SOMObject⟩, the class information of which is in the file ⟨somobj.sc⟩.

4. Data Section

This section declares any data elements contained by objects of this class. By default, data can be accessed only by methods of the class.

5. Methods Section

This section declares methods to which objects of this class can respond. By default, all methods declared in this section are available to any class client. The class definition file, ⟨student.csc⟩, describes a non-derived ⟨Student⟩ class, and is set forth below.

```
static void setUpStudent(
    Student *somSelf, char *id, char *name)
{
    StudentData *somThis =    StudentGetData(somSelf);
    strcpy(_id, id);
    strcpy(_name, name);
}
static void printStudentInfo(Student *somSelf)
{
    StudentData *somThis =    StudentGetData(somSelf);
    printf("    Id          : %s \n", _id);
    printf("    Name        : %s \n", _name);
    printf("    Type        : %s \n", _getStudentType(somSelf));
}
static char *getStudentType(Student *somSelf)
{
    StudentData *somThis =    StudentGetData(somSelf);
    static char *type = "student";
    return (type);
}
static char *getStudentId(Student *somSelf)
{
    StudentData *somThis =    StudentGetData(somSelf);
    return (_id);
}
```

Notice that the method code appears similar to standard C. First, each method takes, as its first parameter, a pointer (⟨somSelf⟩) to the target object. This parameter is implicit in the class definition file, but is made explicit in the method implementation. Second, each method starts with a line setting an internal variable named ⟨somThis⟩, which is used by macros defined within the SOM header file. Third, names of data elements of the target object are preceded by an underscore character "-". The underscored name represents a C language macro defined in the class header file. Fourth, methods are invoked by placing an underscore "-" in front of the method name. This underscored name represents a macro for message resolution and shields a programmer from having to understand the details of this process.

The first parameter of every method is always a pointer to the target object. This is illustrated below in the method ⟨printStudentInfo()⟩ which invokes the method ⟨getStudentType()⟩ on its target object.

⟨**getStudentId()**⟩ from the ⟨**Student**⟩ base class.
The class definition file for ⟨**GraduateStudent**⟩, ⟨**graduate.csc**⟩, is set forth below.

```
Class Definition File:  <graduate.csc>

include <student.sc>

class:
  GraduateStudent;

parent:
  Student;


data:
  char  thesis[128];    /* thesis title */
  char  degree[16];     /* graduate degree type */

methods:
  override  printStudentInfo;
  override  getStudentType;
  void  setUpGraduateStudent(
        char *id, char *name, char *thesis, char *degree);
```

The method implementation file, ⟨graduate.c⟩, is shown below.

```
Class Definition File:  <undgrad.csc>
include <student.sc>

class:
  UnderGraduateStudent;

parent:
  Student;

data:
  char  date[16];     /* graduation date */

methods:
  override  printStudentInfo;
  override  getStudentType;
  void  setUpUnderGraduateStudent(
        char *id, char *name, char *date);
```

The method implementation file, (undgrad.c), is set forth below.

```
Class Method Implementation File:  <undgrad.c>

#define UnderGraduateStudent_Class_Source
#include "undgrad.ih"

static void printStudentInfo(
    UnderGraduateStudent *somSelf)
```

```
    int     credit;             /* number of credits  */
    int     capacity;           /* maximum number of seats */
    Student *studentList[20]; /* enrolled student list */
    int     enrollment;         /* number of enrolled students */

methods:
    override  somInit;

    void  setUpCourse(char *code, char *title,
        char *instructor, int credit, int capacity);
    -- sets up a new course.

    int   addStudent(Student *student);
    -- enrolls a student to the course.

    void  dropStudent(char *studentId);
    -- drops the student from the course.

    void  printCourseInfo();
    -- prints course information.
```

Often classes will want to take special steps to initialize their instance data. An instance of ⟨Course⟩ must at least initialize the ⟨enrollment⟩ data element, to ensure the array index starts in a valid state. The method ⟨somInit()⟩ is always called when a new object is created. This method is inherited from ⟨SOMObject⟩, and can be overridden when object initialization is desired.

This example brings up an interesting characteristic of inheritance, the "is-a" relationship between derived and base classes. Any derived class can be considered as a base class. We say that a derived class "is-a" base class. In the previous example, any ⟨GraduateStudent⟩ "is-a" ⟨Student⟩, and can be used anyplace we are expecting a ⟨Student⟩. The converse is not true. A base class is not a derived class. A ⟨Student⟩ can not be treated unconditionally as a ⟨GraduateStudent⟩. Thus, elements of the array ⟨studentList⟩ can point to either ⟨Student⟩s, a ⟨GraduateStudent⟩s, or a ⟨UnderGraduateStudent⟩s.

The method implementation file for ⟨Course⟩, ⟨course.c⟩, is set forth below.

```
}
static void printCourseInfo(Course *somSelf)
{
    int i;
    CourseData *somThis = CourseGetData(somSelf);
    printf("  %s %s \n", _code, _title);
    printf("  Instructor Name : %s \n",   _instructor);
    printf("  Credit = %d, Capacity = %d, Enrollment = %d \n\n",
             _credit, _capacity, _enrollment);
    printf("  STUDENT LIST: \n\n");
    for(i=0; i<_enrollment; i++) {
      _printStudentInfo(_studentList[i]);
      printf("\n");
    }
}
```

Notice in particular the method ⟨printCourseInfo()⟩. This method goes through the array ⟨studentList⟩ invoking the method ⟨printStudentInfo()⟩ on each student. This method is defined for ⟨Student⟩, and then overridden by both ⟨GraduateStudent⟩ and ⟨UnderGraduateStudent⟩. Since the array element can point to any of these three classes, it is impossible at compile time to determine what the actual type of the target object is, only that the target object is either a ⟨Student⟩ or some type derived from ⟨Student⟩. Since each of these classes defines a different ⟨printStudentInfo()⟩ method, it is impossible to determine which of these methods will be invoked with each pass of the loop. This is all under the control of override resolution.

THE SOM CLIENT

To understand how a client might make use of these four classes in a program, an example is presented below in the file ⟨main.c⟩. The example illuminates object instantiation and creation in SOM, and how methods are invoked.

```
SOM client code:  <main.c>
#include <student.h>
#include <course.h>
#include <graduate.h>
#include <undgrad.h>
main()
```

that are of the same class as this object also contain an address that points to this method procedure table diagrammed at label **240**. Any methods inherited by the objects will have their method procedure addresses at the same offset in memory as they appear in the method procedure table as set forth at label **240** of the ancestor class from which it is inherited.

Addresses of the blocks of computer memory containing the series of instructions for two of the method procedures are set forth at labels **250** and **260**. Labels **270** and **280** represent locations in a computer memory containing the series of instructions of particular method procedures pointed to by the addresses represented by labels **250** and **260**.

The SOM Base Classes

Much of the SOM Object Model is implemented by three classes that are part of the basic SOM support. Briefly these classes are:

SOMObject - This class is the root class of all SOM classes. Any class must be descended from SOMObject. Because all classes are descended from SOMObject they all inherit and therefore support the methods defined by SOMObject. The methods of SOMObject like the methods of any SOM class can be overridden by the classes descended from SOMObject.

SOMClass - This class is the root meta class for all SOM meta classes. A meta class is a class whose instances are class objects. SOMClass provides the methods that allow new class objects to be created.

SOMClassMgr - This class is used to create the single object in a SOM based program that manages class objects.

The three SOM base classes are defined below.

SOMObject

This is the SOM root class, all SOM classes must be descended from ⟨SOMObject⟩. ⟨SOMObject⟩ has no instance data so there is no per-instance cost to being descended from it.

SOMObject has the following methods:

Method: somInit

Parameters: somSelf

Returns: void

Description:

Initialize ⟨self⟩. As instances of ⟨SOMObject⟩ do not have any instance data there is nothing to initialize and you need not call this method. It is provided to induce consistency among subclasses that require initialization.

⟨somInit⟩ is called automatically as a side effect of object creation (ie, by ⟨somNew⟩). If this effect is not desired, you can supply your own version of ⟨somNew⟩ (in a user-written metaclass) which does not invoke ⟨somInit⟩.

When overriding this method you should always call the parent class version of this method BEFORE doing your own initialization.

Method: somUninit

Parameters: somSelf

Returns: void

Description:

(Un-initialize self) As instances of ⟨SOMObject⟩ do not have any instance data there is nothing to un-initialize and you need not call this method. It is provided to induce consistency among subclasses that require un-initialization.

Use this method to clean up anything necessary such as dynamically allocated storage. However this method does not release the actual storage assigned to the object instance. This method is provided as a complement to ⟨somFree⟩ which also releases the storage associated with a dynamically allocated object. Usually you would just call ⟨somFree⟩ which will always call ⟨somUninit⟩. However, in cases where ⟨somRenew⟩ (see the definition of ⟨SOMClass⟩) was used to create an object instance, ⟨somFree⟩ cannot be called and you must call ⟨somUninit⟩ explicitly.

When overriding this method you should always call the parentclass version of this method AFTER doing your own un-initialization.

Method: somFree

Parameters: somSelf

Returns: void

family (such as integer) SOM only supports the largest member.

Method: somDispatchV

Parameters: somSelf,
somId methodId,
somId descriptor, ...
Returns: void
Description:

Does not return a value.

Method: somDispatchL

Parameters: somSelf, somId methodId, somId descriptor
Returns: integer4
Description:

Returns a 4 byte quantity in the normal manner that integer data is returned. This 4 byte quantity can, of course, be something other than an integer.

Method: somDispatchA

Parameters: somSelf, somId methodId, somId descriptor
Returns: void *
Description:

Returns a data structure address in the normal manner that such data is returned.

Method: somDispatchD

Parameters: somSelf, somId methodId, somId descriptor
Returns: float8
Description:

Returns a 8 byte quantity in the normal manner that floating point data is returned.

SOMObject methods that support development

The methods in this group are provided to support program development. They have been defined in such a way that most development contexts will find them easy to exploit. However, some contexts may need to customize their I/O facilities. We have attempted to allow this customization in a very portable manner, however not all contexts will be able to perform the customization operations directly because they require passing function parameters. We chose this approach because it allows great platform-neutral flexibility and we felt that any provider of development support would find it reasonable to provide the customizations necessary for her/his specific development environment.

The chosen approach relies on a character output routine. An external variable, ⟨SOMOutCharRoutine⟩, points to this routine. The SOM environment provides an implementation of this routine that should work in most development environments (it writes to the standard output stream). A development context can, however, assign a new value to ⟨SOMOutCharRoutine⟩ and thereby redefine the output process. SOM provides no special support for doing this assignment.

Method: somPrintSelf

Parameters: somSelf
Returns: SOMAny *
Description:

Uses ⟨SOMOutCharRoutine⟩ to write a brief string with identifying information about this object. The default implementation just gives the object's class name and its address in memory. ⟨self⟩ is returned.

Method: somDumpSelf

Parameters: somSelf, int level
Returns: void
Description:

Uses ⟨SOMOutCharRoutine⟩ to write a detailed description of this object and its current state. ⟨level⟩ indicates the nesting level for describing compound objects it must be greater than or equal to zero. All lines in the description will be preceded by ⟨2*level⟩ spaces.

This routine only actually writes the data that concerns the object as a whole, such as class, and uses ⟨somDumpSelfInt⟩ to describe the object's current state. This approach allows readable descriptions of compound objects to be constructed.

Generally it is not necessary to override this method, if it is overridden it generally must be completely replaced.

Method: somDumpSelfInt

Parameters: somSelf
Returns: Zstring
Description:
Returns this object's class name as a NULL terminated string.

Method: somGetParent
Parameters: somSelf
Returns: SOMClass *
Description:
Returns the parent class of self if one exists and NULL otherwise.

Method: somGetClassData
Parameters: somSelf
Returns: somClassDataStructure *
Description:
Returns a pointer to the static ⟨className⟩ClassData structure.

Method: somSetClassData
Parameters: somSelf, somClassDataStructure *cds
Returns: void
Description:
Sets the class' pointer to the static ⟨className⟩ClassData structure.

Method: somDescendedFrom
Parameters: somSelf, SOMClass *Aclassobj
Returns: int
Description:
Returns 1 (true) if ⟨self⟩ is a descendent class of ⟨Aclassobj⟩ and 0 (false) otherwise. Note: a class object is considered to be descended itself for the purposes of this method.

Method: somCheckVersion
Parameters: somSelf, integer4 majorVersion, integer4 minorVersion
Returns: int
Description:
Returns 1 (true) if the implementation of this class is compatible with the specified major and minor version number and false (0) otherwise. An implementation is compatible with the specified version numbers if it has the same major version number and a minor version number that is equal to or greater than ⟨minorVersion⟩. The major, minor version number pair (0,0) is considered to match any version. This method is usually called immediately after creating the class object to verify that a dynamically loaded class definition is compatible with a using application.

Method: somFindMethod
Parameters: somSelf, somId methodId, somMethodProc **m
Returns: int
Description:
Finds the method procedure associated with ⟨methodId⟩ for this class and sets ⟨m⟩ to it. 1 (true) is returned when the method procedure is directly callable and 0 (false) is returned when the method procedure is a dispatch function.

If the class does not support the specified method then ⟨m⟩ is set to NULL and the return value is meaningless.

Returning a dispatch function does not guarantee that a class supports the specified method; the dispatch may fail.

Method: somFindMethodOk
Parameters: somSelf, somId methodId, SomMethodProc **m
Returns: int
Description:
Just like ⟨somFindMethod⟩ except that if the method is not supported then an error is raised and execution is halted.

Method: somFindSMethod
Parameters: somSelf, somId methodId
Returns: somMethodProc *
Description:
Finds the indicated method, which must be a static method defined for this class, and returns a pointer to its method procedure. If the method is not defined (as a static method or at all) for this class then a

⟨methodDescriptor⟩ is a somId for a string describing the calling sequence to this method as described in ⟨somcGetNthMethodInfo⟩ defined in the SOMObject class definition.

⟨method⟩ is the actual method procedure for this method

⟨redispatchStub⟩ is a procedure with the same calling sequence as ⟨method⟩ that re-dispatches the method to one of this class's dispatch functions.

⟨applyStub⟩ is a procedure that takes a standard variable argument list data structure applies it to its target object by calling ⟨method⟩ with arguments derived from the data structure. Its calling sequence is the same as the calling sequence of the dispatch methods defined in SOMObject. This stub is used in the support of the dispatch methods used in some classes. In classes where the dispatch functions do not need such a function this parameter may be null.

Method: somOverrideSMethod

Parameters: somSelf, somId methodId, somMethodProc *method

Returns: void

Description:

This method can be used instead of ⟨somAddStaticMethod⟩ or ⟨somAddDynamicMethod⟩ when it is known that the class' parent class already supports this method. This call does not require the method descriptor and stub methods that the others do.

Method: somGetMethodOffset

Parameters: somSelf, somId methodId

Returns: integer4

Description:

Returns the specified method's offset in the method procedure table assuming this is a static method, returns 0 if it was not. This value is used to set the offset value in this class data structure. It should only be necessary to use this method when a class used to define a method that it now inherits.

Method: somGetApplyStub

Parameters: somSelf, somId methodId

Returns: somMethodProc *

Description:

Returns the apply stub associated with the specified method. NULL is returned if the method is not supported by this class. An apply stub is a procedure that is called with a fixed calling sequence, namely (SOMAny *self, somId methodId, somId descriptor, ap_list ap) where ⟨ap⟩ is a varargs data structure that contains the actual argument list to be passed to the method. The apply stub forwards the call to its associated method and then returns any result produced by the method.

SOMClassMgr

SOMClassMgr is descended from SOMObject.

SOMObject defines the following methods:

Method: somFindClsInFile

Parameters: somself, somId classId, int majorVersion, int minorVersion, Zstring file

Returns: SOMClass *

Description:

Returns the class object for the specified class. This may result in dynamic loading. If the class already exists ⟨file⟩ is ignored, otherwise it is used to locate and dynamically load the class. Values of 0 for major and minor version numbers bypass version checking.

Method: somFindClass

Parameters: somSelf, somId classId, int majorVersion, int minorVersion

Returns: SOMClass *

Description:

Returns the class object for the specified class. This may result in dynamic loading. Uses somLocateClassFile to obtain the name of the file where the class' code resides, then uses somFindClsInFile.

Method: somClassFromId

Parameters: somSelf, somId classId

Returns: SOMClass *

Description:

Finds the class object, given its Id, if it already exists. Does not load the class. Returns NULL if the class object does not yet exist.

Method: somRegisterClass

representing offsets associated with this class as signified by the elipses and "N + 1" at label **350**. The additional entry is necessary because of the first entry represents a pointer to the class object data structure **248** in Figure 2.

The order of the values in the class data structure is determined by the order of the corresponding method or public instance variable name in the release order section of the class OIDL file. Methods or public data members defined in the class but not mentioned in the release order section are ordered after those mentioned in the release order section and in the order in which they appear in the class OIDL file.

Object Interface Definition Language (OIDL)

The invention redefines language dependent object definitions as a neutral set of information from which object support for any language is provided. The neutral set of information is referred to as an Object Interface Definition Language (OIDL) definition in SOM. SOM OIDL provides the basis for generating binding files that enable programming languages to use and provide SOM objects and their definitions (referred to as classes). Each OIDL file defines the complete interface to a class of SOM objects.

OIDL files come in different forms for different languages. The different forms enable a class implementer to specify additional language-specific information that allows the SOM Compiler to provide support for constructing the class. Each of these different forms share a common core language that specifies the exact information that a user must know to use a class. One of the facilities of the SOM Compiler is the extraction of the common core part of a class definition. Thus, the class implementer can maintain a language-specific OIDL file for a class, and use the SOM Compiler to produce a language-neutral core definition as needed.

This section describes OIDL with the extensions to support C-language programming. As indicated above, OIDL files are compiled by the SOM Compiler to produce a set of language-specific or use-specific binding files.

The SOM Compiler produces seven different files for the C language.

- A public header file for programs that use a class. Use of a class includes creating instance objects of the class, calling methods on instance objects, and subclassing the class to produce new classes.
- A private header file, which provides usage bindings to any private methods the class might have.
- An implementation header file, which provides macros and other material to support the implementation of the class.
- An implementation template, which provides an outline of the class' implementation that the class provider can then edit.
- A language-neutral core definition.
- A private language-neutral core file, which contains private parts of the class interface.
- An OS/2 .DEF file that can be used to package the class in the form of an OS/2 DLL.

OIDL files can contain the following sections:

- Include section;
- Class section;
- Release Order section;
- Metaclass section;
- Parent Class section;
- Passthru section;
- Data section; and
- Methods section.

Include section

This required section contains an include statement that is a directive to the OIDL preprocessor telling the compiler where to find the class interface definition for this class' parent class, the class' metaclass if the class specifies one, and the private interface files for any ancestor class for which this class overrides one or more of its private methods.

Class Section

This required section introduces the class, giving its name, attributes and optionally a description of the class as a whole.

can be packaged in self-contained files or placed in a DLL so the class can be used from many programs.

Referring to Figure **4**, control commences at terminal **400** and flows directly into function block **404** where a SOM language neutral object interface definition (OIDL) **402** is input to the SOM OIDL compiler **404**. The SOM OIDL compiler parses the object definitions in OIDL into a canonical form **406** to simplify the code generation process as input to the target language emitter **410**. The language emitter **410** generates language bindings **414** which include the class data structure depicted in Figure **3**. Control flows to the language compiler shown in function block **420** which receives additional inputs from the language applications **416** and the SOM bindings **412**. The language compiler could be a C, Fortran, Cobol or other compiler depending on user preference. Output from the language compiler is an object file **422** which can be link edited with the SOM runtime library for subsequent execution.

Figure **5** is a flowchart depicting a link, load and execution of an application using SOM objects in accordance with the subject invention. Processing commences at terminal **500** and immediately flows into function block **530** for a dynamic link and load of the SOM objects **510** created in Figure **4** at label **422** and the SOM run time library **520**. Then, at function block **540**, the application is started, which invokes the creation of necessary classes and objects as set forth in function block **550** and detailed in Figures **6**, **7**, **8**, **9** and **10**. Finally, the application is executed as shown in function block **560** and control is terminated at terminal block **570**.

Version Independence For Object Oriented Programs

This aspect of the invention generally relates to improvements in object oriented applications and more particularly solving problems arising from the independent evolution of object definition libraries and the computer applications that use them.

The version independence processing isolates the executable binary form of computer applications that use object definition libraries (also called object class libraries) from certain changes in the implementations or specification of the object definitions that naturally arise during the lifecycle of the libraries. Specifically, the following changes can be made to an object definition without compromising its use by the unmodified executable binary form of a computer application which dynamically loads the object definition each time the application is executed:

1) add new methods to an object definition;

2) move the point of definition for a method from a child class to its parent class;

3) add to, delete from, or otherwise change the private instance data associated with an object definition; and

4) insert a new class definition into a class hierarchy.

This processing is accomplished by the operation of an algorithm in the memory of a processor employing several techniques as follows. Method and instance offset are removed from application binary images. In static object models, such as the one defined in C++, an offset (an integer number) into a method procedure table is used to select a method procedure for each particular method name. The offset depends on the number and order of the methods of the class the method is defined in and the number of methods defined by its ancestors.

This approach has the benefit of being a very fast form of method resolution. However, in the prior art object models have placed these offsets in the binary images of the applications that used a particular object class, resulting in the requirement to recompile the application whenever the offsets required a change.

In SOM, the offsets associated with methods are collected into a single memory data structure for each class, called the class data structure, detailed in the discussion of Figure **3**. This data structure is given an external name and its contents are referred to in applications. Each class data structure is initialized to contain the appropriate offset values when a class object is initialized as detailed in Figure **10**. Thus each time an application is executed all the offset values are recalculated based on the current definitions of the classes used by the application.

Note that any references in an application's binary images to the values stored in the class data structure contain offsets. However, these offsets can remain constant across the four kinds of changes enumerated above. This is because the class data structure only contains offsets for the methods defined in a particular class, not for offsets of methods inherited by the class. Thus, new methods added to a class can have their offsets added at the end of the class data structure without disturbing the positions of the offset values for methods that were already defined in the class.

The SOM Object Interface Definition Language (OIDL) contains a Release Order Section, discussed in the section titled "SOM Object Model" above. The release order section of OIDL allows the class

detected, then the default values of the object are set at function block **850** and control is returned at terminal block **860**.

Figure **9** is a flowchart depicting the detailed initialization of a new SOM class object in accordance with the subject invention. Control commences at terminal **900** and immediately enters a decision block **910** and a test is performed to detect if the parent class of the new SOM class object exists. If a parent class exists, then the parent class is initialized in function block **912**. Once the parent class is initialized, then memory for the class name is allocated at function block **920**. Next, a test is performed again to detect if the parent class of the new SOM class object exists at decision block **930**.

If a parent class does not exist, then initial variables are set to zero as shown in function block **932** and control passes to function block **970**. If a parent class exists, then the initial variables are updated based upon the values from the parent class in function blocks **940**, **950**, and **960**. Then, in function block **970**, the version number for the class is set and error processing is performed in decision block **980**. If an error is detected, then an appropriate message is displayed at output block **982** and processing terminates at terminal block **984**. If no error is detected, then control is returned at terminal block **990**.

Figure **10** is a flowchart depicting the detailed initialization of a SOM class data structure with offset values in accordance with the subject invention. Control commences at terminal block **1000** and immediately flows into function block **1010** where a loop commences with the acquisition of the next static method. In function block **1020**, the new method id is registered with the SOM runtime environment. Then, a test is performed to determine if the method has already been registered in a parent class in decision block **1030**. If the method has been registered, then the method offset is overridden at function block **1032** and control passes to decision block **1070**.

If the method has not been registered with any parent class, then a test is performed to determine if the method has been defined in the current class at decision block **1040**. If the method has been defined, then the existing offsets are employed at function block **1042** and control is passed to decision block **1070**. If the method has not been defined, then memory is allocated and values are initialized in function blocks **1050** and **1060**. In function block **1060** the offset is calculated by adding the number of inherited static methods to the number of inherited static methods processed to date by the class. Error processing is performed in decision block **1070**, and if an error is detected, then an appropriate message is displayed at output block **1072** and processing terminates at terminal block **1074**. After error processing is completed, another test is performed at decision block **1080** to determine if any additional methods require processing. If there are additional methods, then control passes to function block **1010** for the next iteration of the loop. Otherwise, control flows to terminal **1090** where control returns

Parent Class Shadowing

Logic for providing a dynamic insertion of a replacement parent class, referred to in object programming as a parent class shadow, is detailed in this section of the invention. This processing allows the statically compiled definition of what parent class is linked to a particular class at runtime to be dynamically altered during execution. The ability to insert a new parent class into a statically compiled class hierarchy offers more flexibility to maintain and enhance existing code after it has appeared in binary form. It also offers a new degree of freedom for customizing code without access to source materials since this result can be achieved without recompilation.

Prior art systems have inherent limitations associated with statically linking derived classes and their parent classes. These limitations include, computation of the size of the derived object state data structure, initialization of the derived method procedure table, and the inability to provide access to a parent class' methods from within the derived class' methods (called parent class resolution).

The SOM object model removes these static references by having all the parent class information available at runtime through the parent class object. Thus, when the derived class implementation needs information about the size of the parent class' state data structure, the addresses of the parent class' method procedures, or access to the parent class' method procedure table (to support parent class resolution) an appropriate call is placed to acquire the information from the parent class object. The detailed processing to obtain this information are given in Figures **7**, **8**, **9**, and **10**.

SOM introduces a class manager for every SOM process. The class manager is responsible for keeping a registry of classes. The class construction code generated by the SOM compiler works with the class manager to establish the relationship between a class and its parent class whenever a child class object is created. The SOM class manager is an instance of a class which can be subclassed like any other SOM class.

that can be placed into a table of procedure entry points. The table of procedure entry points are used in a static object model as a substitute for the actual method entry point that is expected. The redispatch stub is generated automatically based on the requirements of the dynamic object model. The redispatch stub converts the call generated in the static object model into the form necessary in the dynamic object model and supplies any missing information in the process. Thus, if an object is accessed from a static object model that is provided by a dynamic object model, it can be represented to the static object model via a table of entry points which each indicate a particular redispatch stub.

Figure **12** is a flow diagram depicting the redispatch method in accordance with the subject invention. Label **1200** is a state data structure for a particular object. The first full word at label **1210** contains the address of the object's method procedure table label **1240**. The rest of the state data structure is set forth at label **1230** contains additional information pertaining to the object. The method procedure table set forth at label **1240** containing the addresses of various methods for the particular object. All objects that are of the same class as this object also contain an address that points to this method procedure table diagrammed at label **1240**. Any methods inherited by the objects will have their method procedure addresses at the same offset in memory as they appear in the method procedure table as set forth at label **1240** of the ancestor class from which it is inherited.

In the figure, label **1250** contains a pointer to a redispatch stub **1270**. A redispatch stub is a sequence of instructions that appear as a method to a client program. However, the instructions merely convert the method call into a call to an object's appropriate dispatch function as illustrated at label **1260**. The address at label **1260** is a pointer to the object's dispatch function **1280**. All SOM objects have a dispatch function. The dispatch function **1280** implements an algorithm to select a particular method based on the parameters passed by the redispatch stub. These parameters include the method's identifier, a string describing a set of arguments passed to the identified method, and a data structure containing the set of arguments.

Offset Values

Figure **13** is a flowchart depicting the detailed initialization of the offset value in a SOM class data structure for a single public instance variable. This logic sequence is repeated for each public instance variable defined in a particular class (see the discussion of the OIDL Data Section above). Control commences at the terminal block **1300** and immediately flows into the function block **1310** where the offset of the instance variable is calculated by adding the instance variable's offset within this class' object state data to the offset of the beginning of this class' object state data within the object state data structure set forth in Figure **2** at label **230**.

The beginning of the class' object state data is determined by adding up the sizes of each of this class' ancestor classes' object state data. Control then passes to function block **1320** when the calculated offset is stored into the position in the class data structure as determined by the position of the public instance variable's name in the OIDL files Release Order Section (see the OIDL Release Order section above and Figure **3** above). Control then flows to the terminal block **1330** and the process is complete.

Redispatch Stubs

Figure **14** is a flowchart depicting the detailed control flow that occurs when a redispatch stub is employed to convert a static method call into a dynamic method call. Control commences at the terminal block **1400** and immediately flows into the function block **1410** where the address of the redispatch stub is determined in the normal static method resolution manner by getting the address stored in the object's method procedure table at an offset contained in the appropriate class data structure at position determined when the class was defined.

Control then passes to function block **1420** where the redispatch stub is called exactly like it was the real static method procedure. Function block **1430** depicts how the redispatch stub calls the object's dispatch method (using normal method resolution as described above). The redispatch stub adds the method's identifier and descriptor to the call as required by the object's dispatch method. These values are incorporated into the redispatch function definition when it is generated by the SOM OIDL compiler. (Note: as detailed in the definition of the SOMObject class above, all classes must support dispatch methods). The object's dispatch method procedure determines which actual method procedure should be called using an algorithm specific to the object's class as shown in function block **1440**.

SOM provides a default implementation of such an algorithm that looks the method's identifier up in a table contained in the object's class object to determine the address of a method procedure. Other object models might use other algorithms. Control then passes to function block **1450** where the method

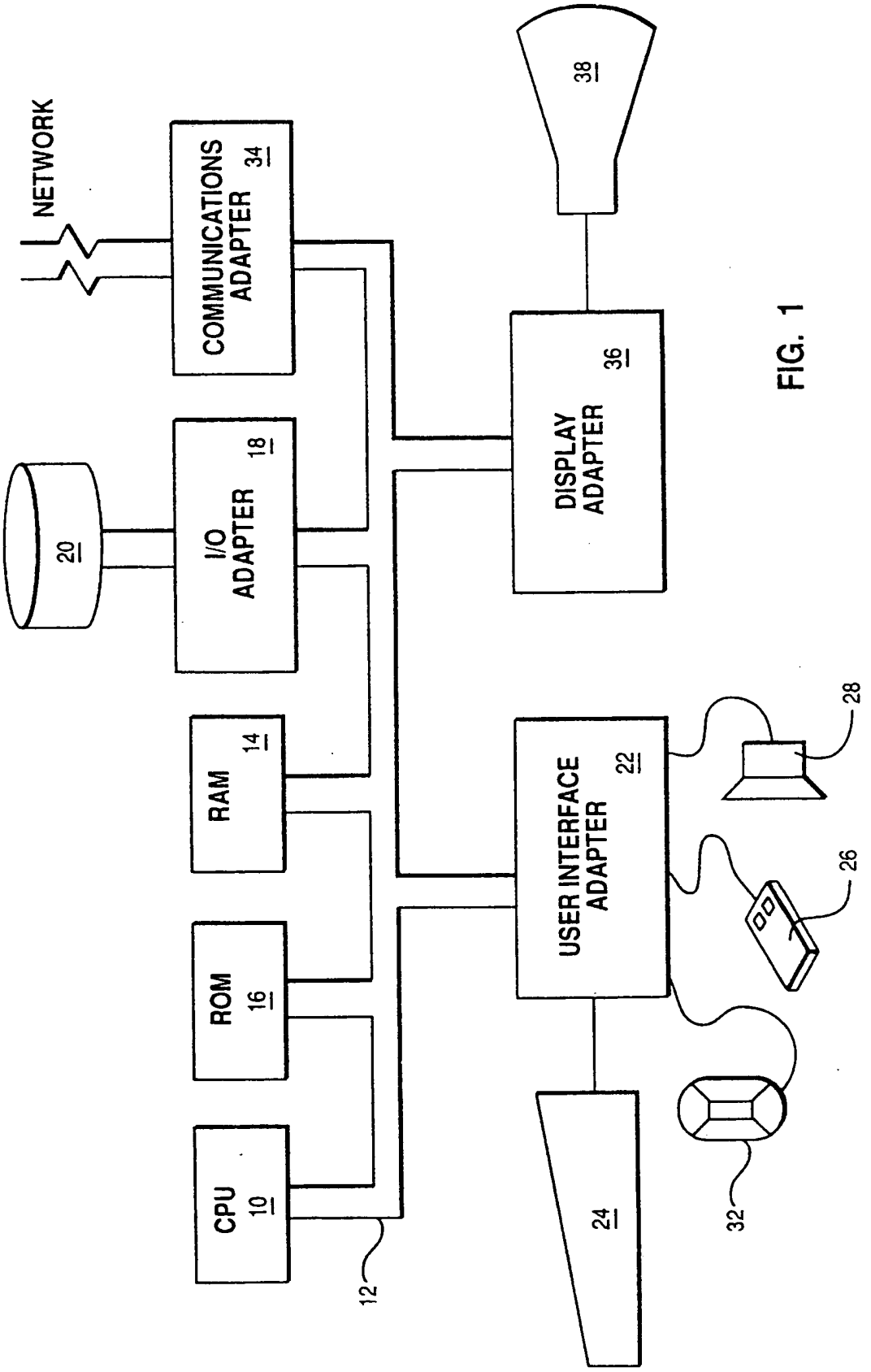
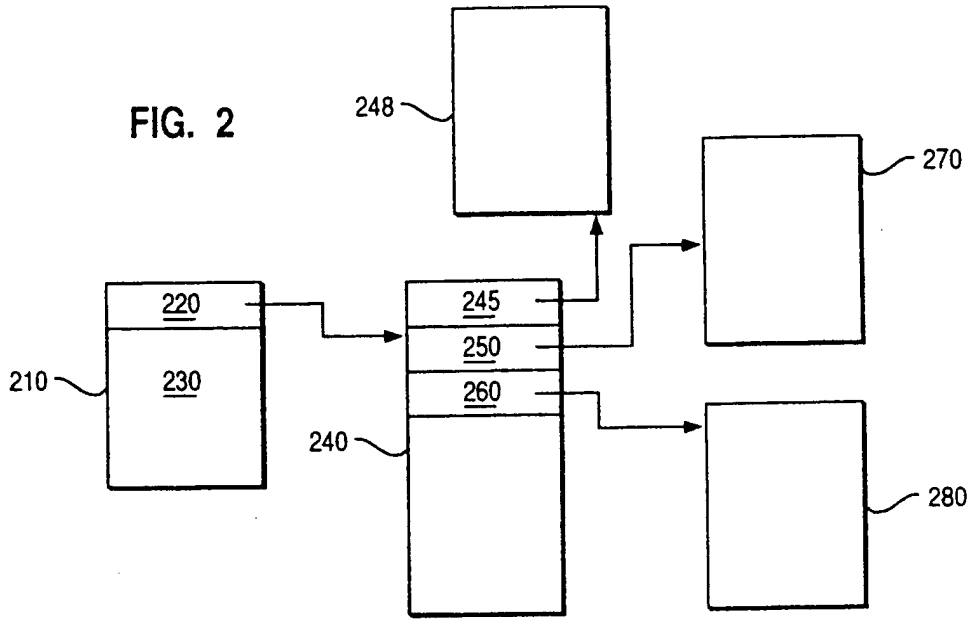

NETWORK
COMMUNICATIONS ADAPTER 34
38
DISPLAY ADAPTER 36
20
I/O ADAPTER 18
RAM 14
ROM 16
CPU 10
12
USER INTERFACE ADAPTER 22
28
26
32
24

FIG. 1

FIG. 2



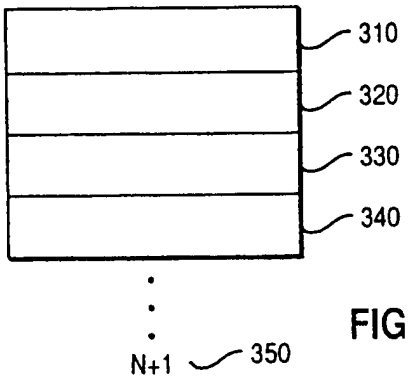


FIG. 3

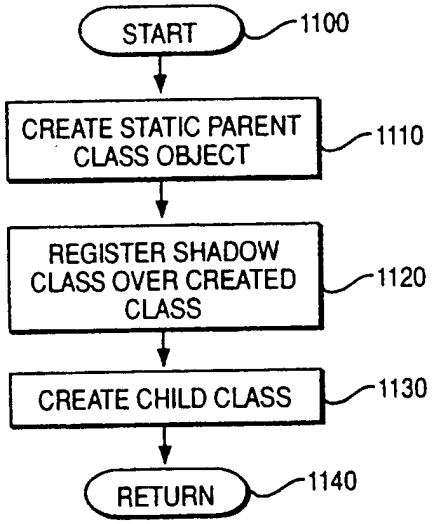


FIG. 11

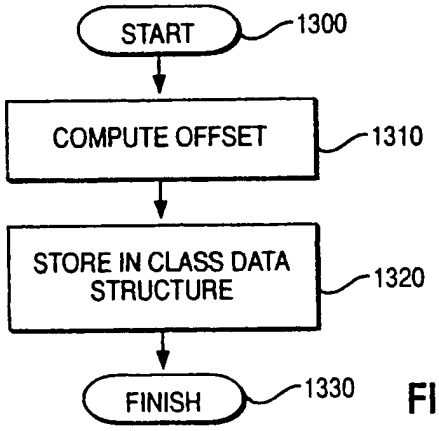


FIG. 13

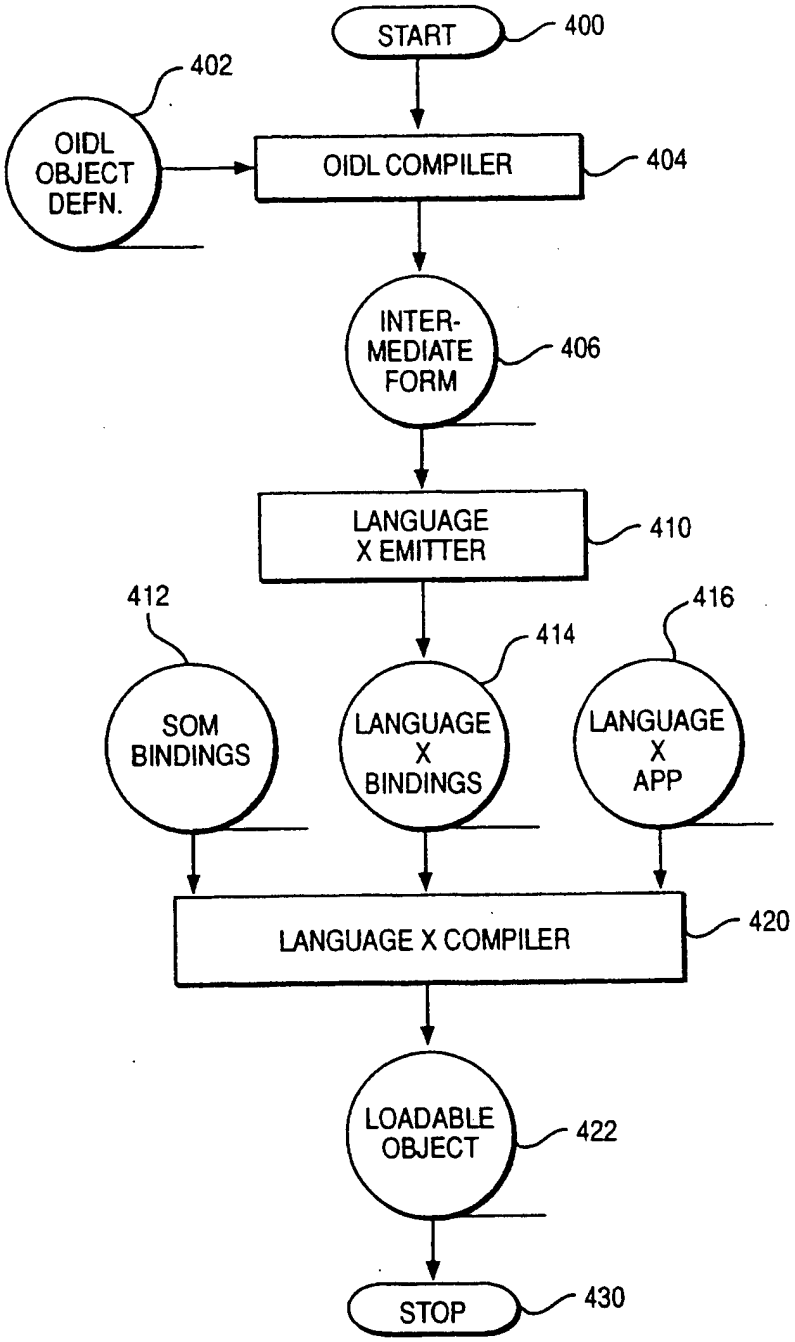

START
400
402
OIDL OBJECT DEFN.
OIDL COMPILER
404
INTER-MEDIATE FORM
406
LANGUAGE X EMITTER
410
412
414
416
SOM BINDINGS
LANGUAGE X BINDINGS
LANGUAGE X APP
LANGUAGE X COMPILER
420
LOADABLE OBJECT
422
STOP
430

FIG. 4

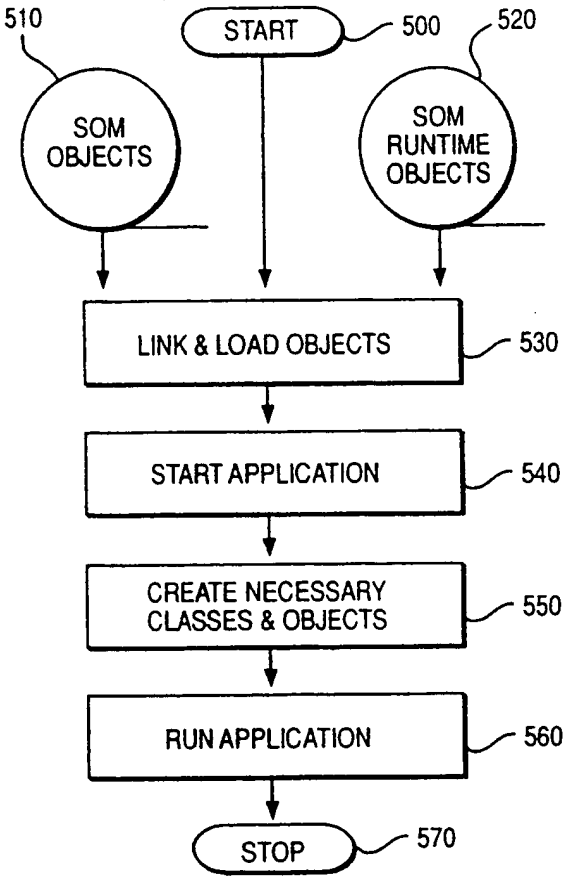


FIG. 5

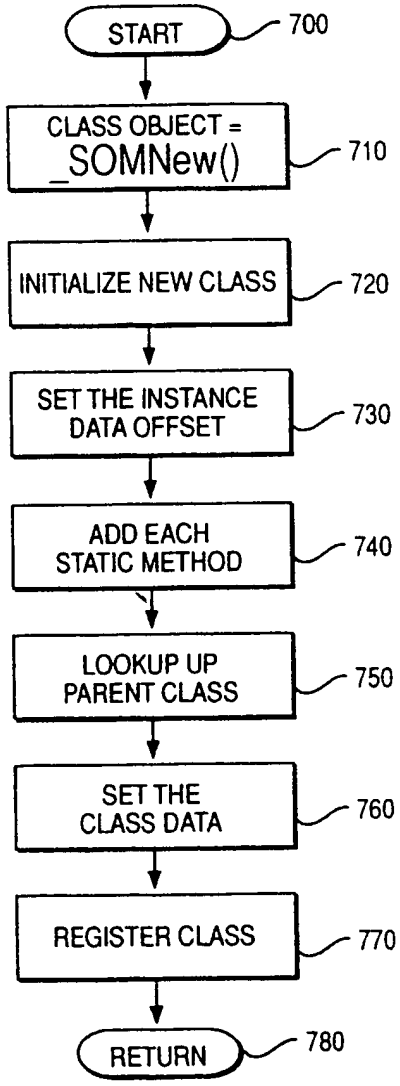


FIG. 7

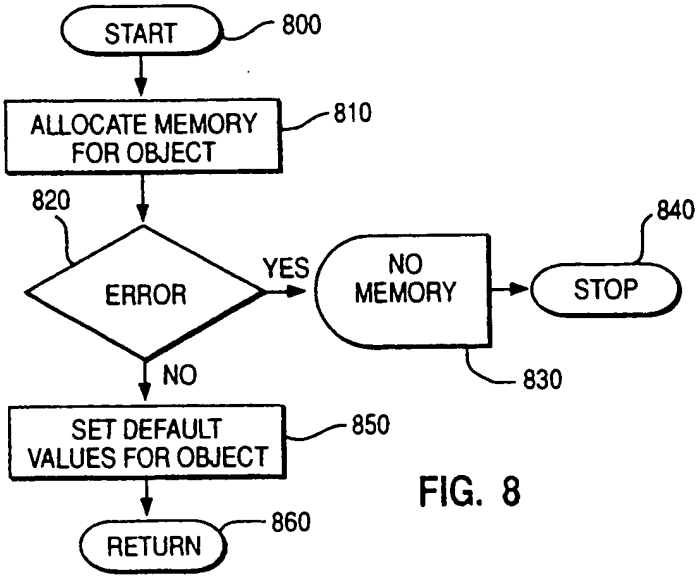


FIG. 8

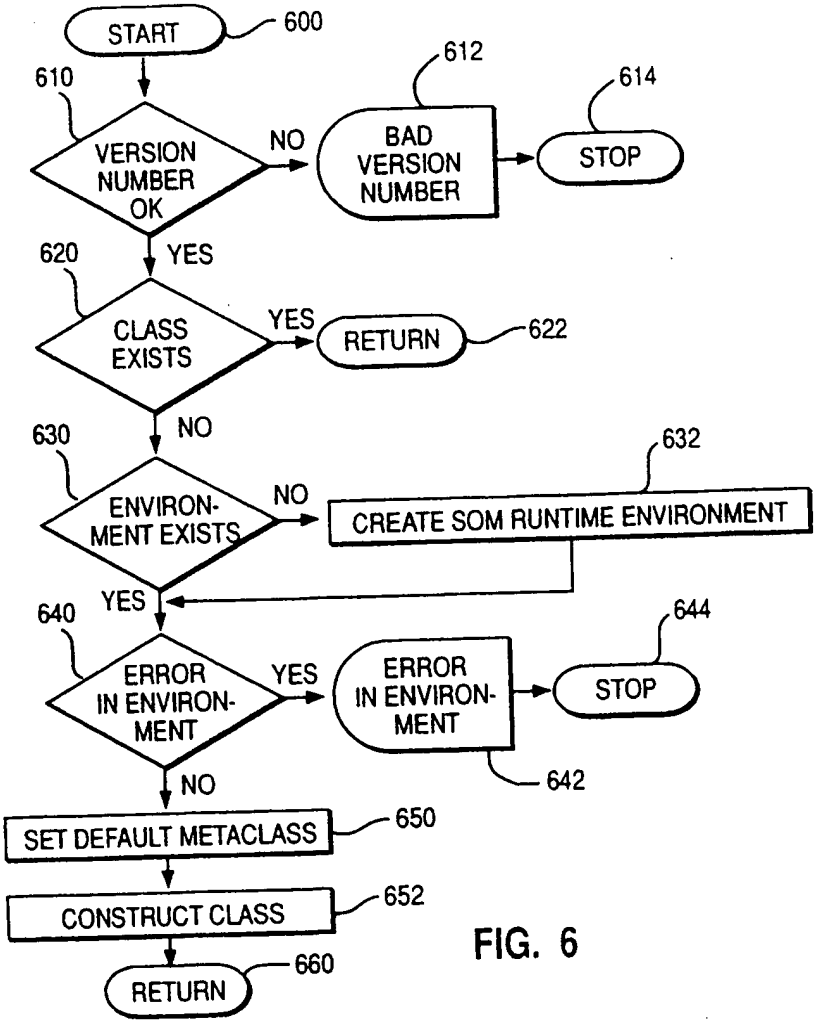

START
600
610
VERSION NUMBER OK
NO
612
BAD VERSION NUMBER
614
STOP
YES
620
CLASS EXISTS
YES
RETURN
622
NO
630
ENVIRON-MENT EXISTS
NO
632
CREATE SOM RUNTIME ENVIRONMENT
YES
640
ERROR IN ENVIRON-MENT
YES
ERROR IN ENVIRON-MENT
644
STOP
642
NO
SET DEFAULT METACLASS
650
CONSTRUCT CLASS
652
RETURN
660

FIG. 6

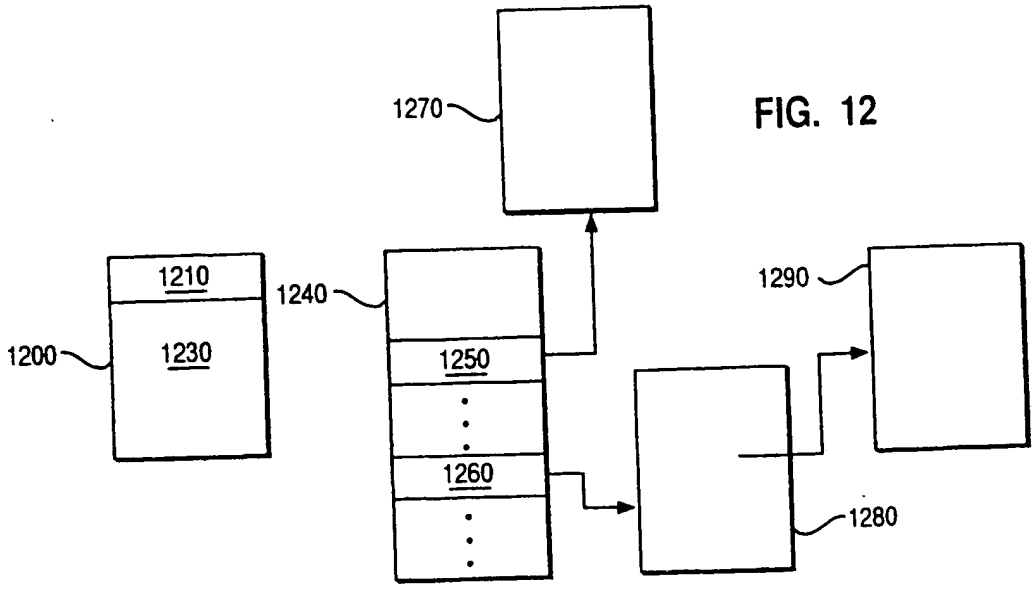

1270
1210
1230
1200
1240
1250
1260
1290
1280

FIG. 12

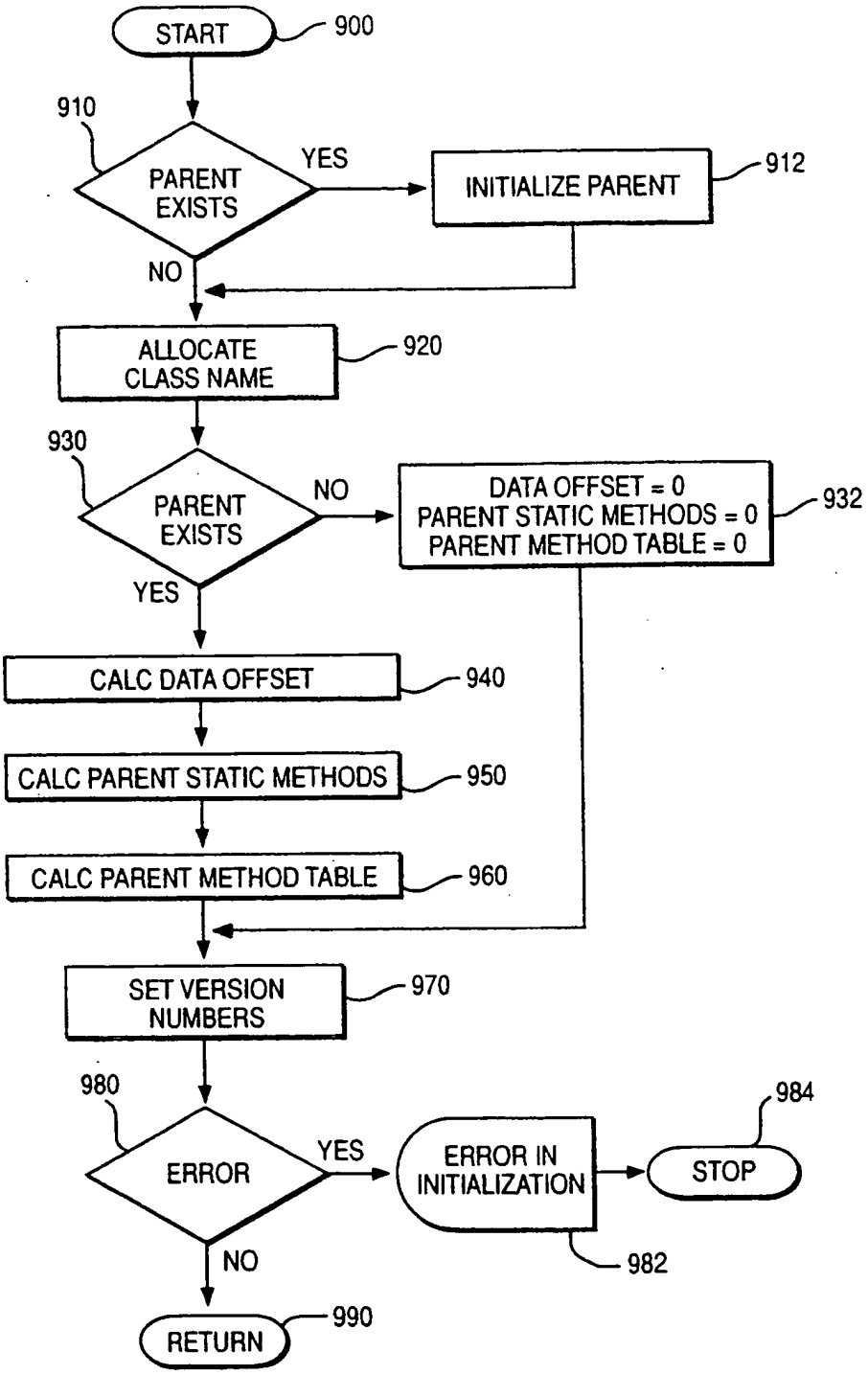

START
900
910
PARENT EXISTS
YES
INITIALIZE PARENT
912
NO
ALLOCATE CLASS NAME
920
930
PARENT EXISTS
NO
DATA OFFSET = 0
PARENT STATIC METHODS = 0
PARENT METHOD TABLE = 0
932
YES
CALC DATA OFFSET
940
CALC PARENT STATIC METHODS
950
CALC PARENT METHOD TABLE
960
SET VERSION NUMBERS
970
980
ERROR
YES
ERROR IN INITIALIZATION
984
STOP
982
NO
RETURN
990

FIG. 9

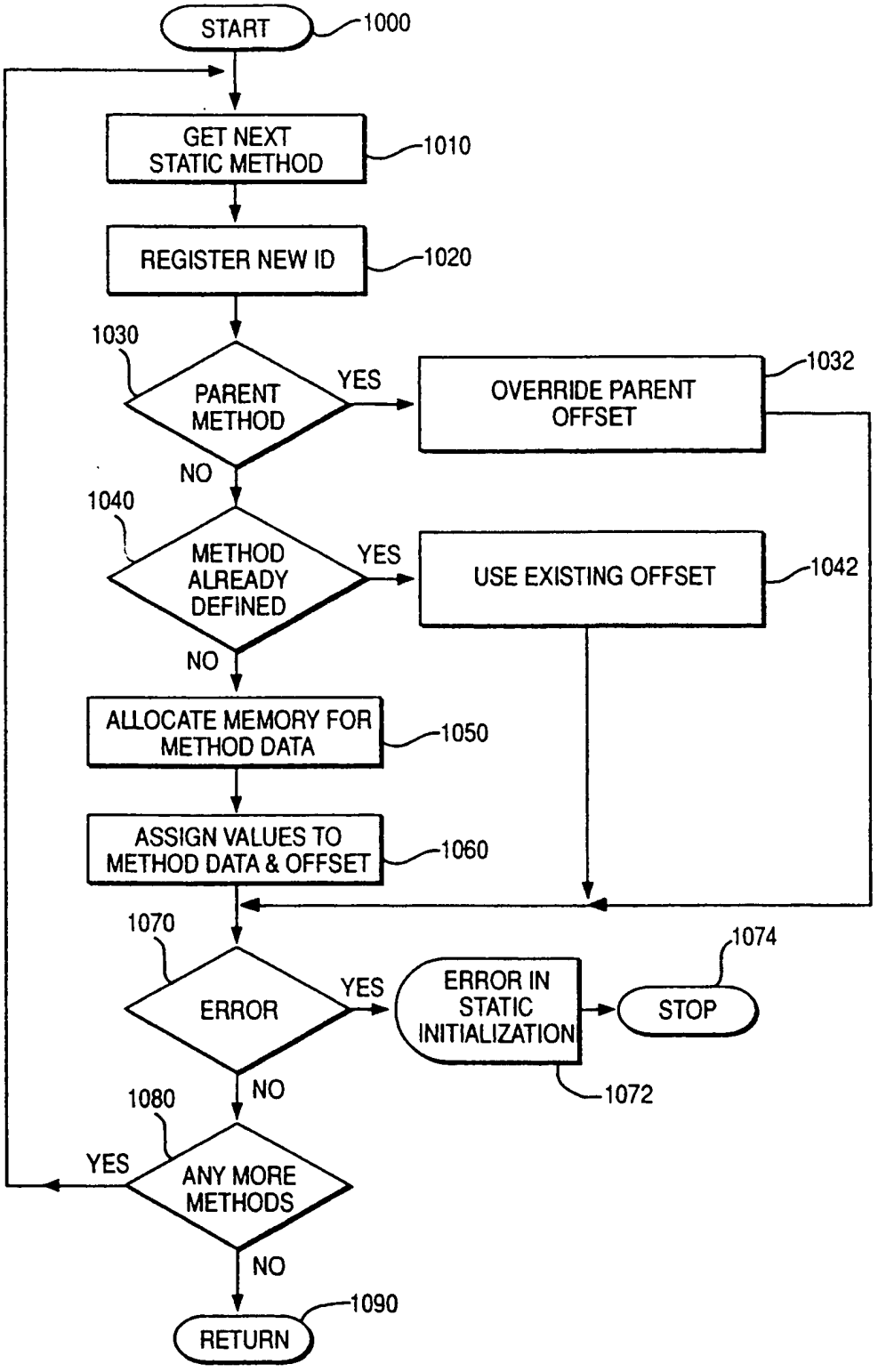

START
1000
GET NEXT STATIC METHOD
1010
REGISTER NEW ID
1020
1030
PARENT METHOD
YES
OVERRIDE PARENT OFFSET
1032
NO
1040
METHOD ALREADY DEFINED
YES
USE EXISTING OFFSET
1042
NO
ALLOCATE MEMORY FOR METHOD DATA
1050
ASSIGN VALUES TO METHOD DATA & OFFSET
1060
1070
ERROR
YES
ERROR IN STATIC INITIALIZATION
1074
STOP
1072
NO
1080
ANY MORE METHODS
YES
NO
1090
RETURN

FIG. 10

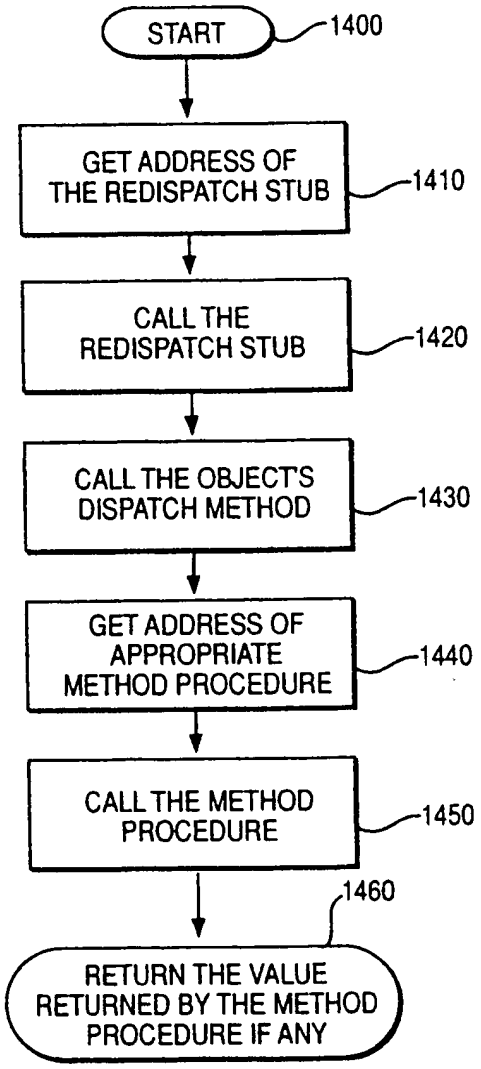

START
1400
GET ADDRESS OF THE REDISPATCH STUB
1410
CALL THE REDISPATCH STUB
1420
CALL THE OBJECT'S DISPATCH METHOD
1430
GET ADDRESS OF APPROPRIATE METHOD PROCEDURE
1440
CALL THE METHOD PROCEDURE
1450
1460
RETURN THE VALUE RETURNED BY THE METHOD PROCEDURE IF ANY

FIG. 14

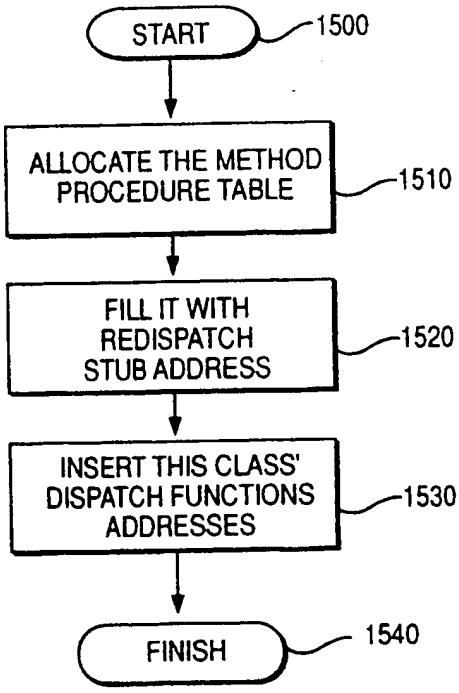

START
1500
ALLOCATE THE METHOD PROCEDURE TABLE
1510
FILL IT WITH REDISPATCH STUB ADDRESS
1520
INSERT THIS CLASS' DISPATCH FUNCTIONS ADDRESSES
1530
FINISH
1540

FIG. 15

(11) Publication number: **0 546 682 A3**

(12) EUROPEAN PATENT APPLICATION

(21) Application number: **92310240.4**

(51) Int. Cl.[5]: **G06F 9/44**

(22) Date of filing: **09.11.92**

(30) Priority: **12.12.91 US 805777**

(43) Date of publication of application:
16.06.93 Bulletin 93/24

(84) Designated Contracting States:
DE FR GB

(88) Date of deferred publication of the search report:
08.12.93 Bulletin 93/49

(71) Applicant: **International Business Machines Corporation**
Old Orchard Road
Armonk, N.Y. 10504(US)

(72) Inventor: **Conner, Mike Haden**
4416 Walhill Lane
Austin, Texas 78759(US)
Inventor: **Martin, Andrew Richard**
1070 Mearns Meadow No. 534
Austin, Texas 78758(US)
Inventor: **Raper, Larry Keith**
7860 Lakewood Drive
Austin, Texas 78750(US)

(74) Representative: **Burt, Roger James, Dr.**
IBM United Kingdom Limited
Intellectual Property Department
Hursley Park
Winchester Hampshire SO21 2JN (GB)

(54) Parent class shadowing.

(57) A method, system and program for supporting a dynamic bind between a derived class and its parent class. A processor provides for the registration of class objects and dynamic binding of derived class objects to their parent class objects based on the registration mechanism. The SOM object model removes static references to class objects by having all the parent class information available at runtime through the parent class object. Thus, when the derived class implementation needs information about the size of the parent class state data structure, the addresses of the parent class method procedures, or access to the parent class method procedure table the appropriate information is retrieved from the parent class object.

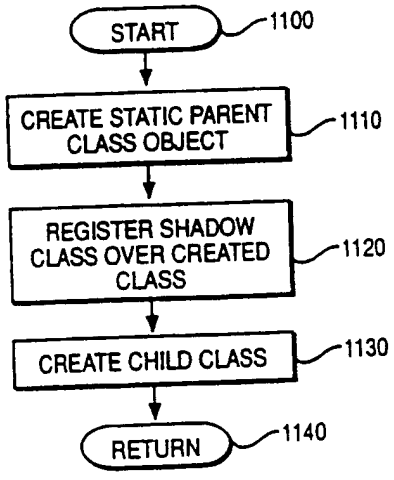


FIG. 11

EP 0 546 682 A3

European Patent Office

EUROPEAN SEARCH REPORT

Application Number

EP 92 31 0240

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int. Cl.5)
X	IBM TECHNICAL DISCLOSURE BULLETIN. vol. 33, no. 10A, March 1991, ARMONK, NY, US page 311 'DYNAMIC REASSOCIATION OF A SUBCLASS WITH A SUPERCLASS' * the whole document *	3-6	G06F9/44
Y	IDEM	1	

X	IBM TECHNICAL DISCLOSURE BULLETIN. vol. 33, no. 10A, March 1991, ARMONK, NY, US page 317 'DYNAMIC REASSOCIATION OF SUBCLASSES WITH A NEW SUPERCLASS' * the whole document *	3-6	
Y	IDEM	1	

X	SYSTEMS & COMPUTERS IN JAPAN. vol. 21, no. 5, 1990, NEW YORK, NY, US pages 1 - 14 R. ONAI ET AL. 'PROPOSAL AND EVALUATION OF DYNAMIC OBJECT-ORIENTED PROGRAMMING' * page 3, right column, line 1 - line 24 * * page 4, right column, line 9 - line 27 * * page 5, right column, line 42 - page 6, left column, line 22 * * page 10, left column, line 34 * * figures 3,8,9; table 2 *	3-6	TECHNICAL FIELDS SEARCHED (Int. Cl.5) G06F
Y	IDEM	1	

Y	GB-A-2 242 293 (APPLE COMPUTER) * abstract * * page 10, line 1 - line 17 * * page 12, line 6 - page 13, line 2 * * page 15, line 5 - page 17, line 2 * * figure 5 *	1	
A	IDEM	3-6	

	-/--		

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	13 OCTOBER 1993	JONASSON J.T.

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P0401)

European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 92 31 0240

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int. Cl.5)
A	OOPSLA '89, CONFERENCE PROCEEDINGS, SPECIAL ISSUE OF SIGPLAN NOTICES. vol. 24, no. 10, October 1989, pages 397 - 406 O.L. MADSEN ET AL. 'VIRTUAL CLASSES A POWERFUL MECHANISM IN OBJECT-ORIENTED PROGRAMMING' * page 398, left column, line 13 - line 33 * * page 405, left column, line 16 - right column, line 14 * * figures 16,17 * -----	1,3-6	

TECHNICAL FIELDS SEARCHED (Int. Cl.5)

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	13 OCTOBER 1993	JONASSON J.T.

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P0401)